A Modular Way to Reason About Iteration

Jean-Christophe Filliâtre, Mário Pereira

A Modular Way to Reason About Iteration

Jean-Christophe Filliâtre[1,2] and Mário Pereira[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. In this paper we present an approach to specify programs performing iterations. The idea is to specify iteration in terms of the finite sequence of the elements enumerated so far, and only those. In particular, we are able to deal with non-deterministic and possibly infinite iteration. We show how to cope with the issue of an iteration no longer being consistent with mutable data.
We validate our proposal using the deductive verification tool Why3 and two iteration paradigms, namely cursors and higher-order iterators. For each paradigm, we verify several implementations of iterators and client code. This is done in a modular way, *i.e.*, the client code only relies on the specification of the iteration.

1 Introduction

Iteration is a central concept in programming. It can be as simple as a while loop or a recursive function, but it can also appear as a more complex artifact, such as a cursor, a higher-order iterator, a generator, or a lazy list. When it comes to verifying the correctness of a program, we need tools to reason about iteration. Typically, we provide a suitable loop invariant for a while loop and a contract for a recursive function. In this paper, we consider the problem of verifying programs where iteration is performed by other means, such as cursors or higher-order iterators. In particular, we are interested in answering the following challenges:

- Iteration is not necessarily the traversal of a data structure. It can be, for instance, the result of an algorithm, such as the enumeration of all prime numbers.
- Iteration is not necessarily finite, as in the aforementioned case of prime numbers.
- Iteration is not necessarily deterministic. The simplest example is that of a symbol generator. From the client point of view, the only required property is that the next element is distinct from the previous ones. Another example is the traversal of a set where elements are presented in some unspecified order. When the iteration is deterministic, however, we want to be able to specify it.

This research was partly supported by the Portuguese Foundation for Sciences and Technology (grant FCT-SFRH/BD/99432/2014) and by the French National Research Organization (project VOCAL ANR-15-CE25-008).

– When iteration depends on mutable data, client code may put iteration in some inconsistent state. In Java, for instance, this problem is solved by maintaining version numbers and by raising an exception in the case of a concurrent modification. In our case, we wish instead to be able to prove, statically, that there is no concurrent modification.
– When a data structure is abstract (for example, a set for which we do not know the implementation) we still want to be able to specify an iteration over its elements and to verify a program using such an iteration. Even when we have access to the implementation of the iteration, we are still interested in performing verification in a modular way with an abstraction barrier. It means verifying the client code independently of a particular implementation for the iteration.

In this paper we propose a way to specify iteration that fulfills all the above-mentioned requirements. We validate our work using the deductive verification tool Why3 [1], but the idea is broader and could be implemented in any other deductive verification tool. Our contribution is twofold:

– An approach to *specify* an iteration process, independently of how it is implemented (cursor, higher-order function, etc.);
– A methodology to *verify* implementations and use of cursors and higher-order iteration functions.

This paper is organized as follows. Sec. 2 introduces our proposal to specify an iteration. Sec. 3 gives a brief overview of Why3. Then we consider cursors in Sec. 4 and higher-order iterators in Sec. 5. We discuss related work in Sec. 6 before concluding. The Why3 developments from this paper can be found at the following address: `http://www.lri.fr/~mpereira/iteration/`.

2 Specifying Iteration

We present in this section our proposal to formally specify an iteration. We use several examples to illustrate this approach, including cases of non-deterministic and infinite iteration.

The idea is to specify the iteration in terms of the finite sequence v of the elements enumerated so far, and only those. More precisely, such a specification is composed of two predicates: the first predicate, called *enumerated*, characterizes the elements of v; the second predicate, called *completed*, indicates whether the iteration is completed. In the following, $\|v\|$ denotes the length of v, $v[i]$ denotes the i-th element of v (assuming a 0-based indexation), and $x \in v$ means that x occurs in v.

Consider for instance the iteration over an array a, from left to right. The first predicate, *enumerated*, is as follows:

$$\mathit{enumerated}(v, a) \triangleq \forall i.\ 0 \leq i < \|v\| \implies v[i] = a[i]$$

In other words, the sequence v is a prefix of the array a. The second predicate, *completed*, simply compares the length of v with that of a:

$$\mathit{completed}(v, a) \triangleq \|v\| = \mathit{length}(a)$$

Let us now consider the iteration over the elements of a finite set s, in a non-deterministic way. Such an iteration can be specified as follows:

$$\begin{aligned} \mathit{enumerated}(v, s) &\triangleq \mathit{distinct}(v) \wedge \forall x.\ x \in v \implies x \in s \\ \mathit{completed}(v, s) &\triangleq \|v\| = \mathit{card}(s) \end{aligned}$$

The condition $\mathit{distinct}(v)$ means that the sequence v contains no duplicate elements, to account for the fact that no element is visited twice (s is a set, not a multiset). We also require the elements of v to be elements of s. Since we do not require any additional property, we have a non-deterministic iteration. The iteration is completed whenever the length of v is equal to the cardinal of s.

Let us now assume that we want to specify instead a *deterministic* iteration over the elements of s. One way to do this is to introduce some oracle function *elements* that returns a sequence containing the elements of s in the order they will be visited. Then *enumerated* merely says that we have already visited a prefix of this sequence, that is,

$$\mathit{enumerated}(v, s) \triangleq \mathit{prefix}(v, \mathit{elements}(s))$$

with a natural definition for *prefix*:

$$\mathit{prefix}(s_1, s_2) \triangleq \|s_1\| \leq \|s_2\| \wedge \forall i.\ 0 \leq i < \|s_1\| \implies s_1[i] = s_2[i]$$

With this specification, the behavior of the enumeration is determined from the beginning. For instance, if the elements of s are totally ordered, then $\mathit{elements}(s)$ could be the sorted sequence of the elements of s.

Let us switch now to examples of iteration that are not traversals of a data structure. Consider for instance an iteration obtained by the repeated application of a function f starting with some initial value x_0, that is, the infinite sequence

$$x_0,\ f(x_0),\ f(f(x_0)),\ f(f(f(x_0))),\ \ldots$$

On way to specify it is as follows:

$$\mathit{enumerated}(v, x_0, f) \triangleq \forall i.\ 0 \leq i < \|v\| \implies v[i] = f^i(x_0)$$

assuming f^i is defined as the ith functional power of f. Besides, to account for the fact that this iteration never halts, we simply define

$$\mathit{completed}(v, x_0, f) \triangleq \mathit{false}$$

The next example is the specification of a scanner for a possibly infinite channel c. The elements of v are characters and a special character `EOF` marks the end of the channel. The specification looks like:

$$\begin{aligned} \mathit{enumerated}(v, c) &\triangleq \cdots \wedge \forall i.\ 0 \leq i < \|v\| - 1 \implies v[i] \neq \texttt{EOF} \\ \mathit{completed}(v, c) &\triangleq \|v\| > 0 \wedge v[\|v\| - 1] = \texttt{EOF} \end{aligned}$$

```
type seq 'a
function length (seq 'a) : int
axiom length_nonnegative: forall s: seq 'a. 0 ≤ length s
constant empty: seq 'a
axiom empty_length: length empty = 0
function ([]) (seq 'a) int : 'a
function snoc (seq 'a) 'a : seq 'a
axiom snoc_length: forall s: seq 'a, x: 'a. length (snoc s x) = 1 + length s
axiom snoc_get:
  forall s: seq 'a, x: 'a, i: int. 0 ≤ i ≤ length s →
  (snoc s x)[i] = if i < length s then s[i] else x
```

Fig. 1: Sequence theory (excerpt).

This specification covers both the case of a finite channel, with a terminal `EOF`, and the case of an infinite channel, where `EOF` never shows up.

Our last example is that of a *symbol generator*, that is, a program that generates fresh symbols on demand. Its output is an infinite iteration of distinct symbols, that is

$$\begin{aligned} \mathit{enumerated}(v) &\triangleq \mathit{distinct}(v) \\ \mathit{completed}(v) &\triangleq \mathit{false} \end{aligned}$$

In this case, *enumerated* does not depend on any information other than the sequence v itself.

3 Why3 in a Nutshell

Our goal is to apply the idea of specifying an iteration using the predicates *enumerated* and *completed* in the context of deductive program verification. To this end, we used the Why3 tool to explore this approach. However, this proposal is general and is not tied to Why3. Any other deductive verification tool could be used. In this section, we briefly describe the Why3 platform, its organization and principal features.

The Why3 platform proposes a set of tools allowing the user to implement, formally specify, and prove programs. The use of Why3 is oriented towards automatic proofs, as it supports many external automatic theorem provers. Why3 can also interact with interactive proof assistants, such as Coq, Isabelle, or PVS, when a proof obligation cannot be automatically discharged.

Why3 comes with a programming language, WhyML [9], an ML dialect with some restrictions in order to make automatic proof simpler. This language offers some features commonly found in functional languages, like pattern-matching, algebraic types and polymorphism, but also imperative constructions, like records with mutable fields and exceptions. Programs written in WhyML can be annotated with contracts, that is, pre- and postconditions. The code itself can be annotated, for instance, to express loop invariants or to justify termination of loops and recursive functions. It is also possible to add intermediate assertions

in the code to ease automatic proofs. The WhyML language allows to write ghost code [8], which is used only for specification and proof purposes and can be removed with no observable modification in the program's execution. The system uses the annotations to generate proof obligations thanks to a weakest precondition calculus.

The logic used to write formal specifications is an extension of first-order logic with rank-1 polymorphic types, algebraic types, (co-)inductive predicates and recursive definitions [7], as well as a limited form of higher-order logic [4]. This logic is used to write theories for the purpose of modeling the behavior of programs. Such theories are most of the time axiomatic. Figure 1 represents a fragment from the sequence theory provided by the Why3 standard library. We can find there the polymorphic type of finite sequences (`seq 'a`), a constant representing the empty sequence (`empty`), function symbols (`length` for the sequence length, $\cdot$`[·]` to access the i-th element, and `snoc` to add an element at the end of a sequence), together with axioms defining these symbols. Why3 standard library is formed of many logic theories of this kind, in particular for integer and floating point arithmetic, sets, and dictionaries.

The entire standard library, numerous verified examples, as well as a more detailed presentation of Why3 and WhyML are available on the project web site, `http://why3.lri.fr`. However, the rest of this paper does not assume any further knowledge of Why3.

4 Cursors

A cursor [5] is a data structure that implements iteration via a function, say `next`, that is called each time we need to get the next element, if there is one. It is thus an iteration paradigm where the control is given to the *consumer*, which calls `next` whenever needed, contrary to other paradigms where control is given to the *producer* of the iteration. Cursors are broadly used in C++ and Java, for instance.

We adopt a model where we interact with the cursor via two functions: `has_next` returns a Boolean indicating the existence of a next element in the iteration; and `next` advances to the next element and returns it. The latter operation updates the cursor by a side effect[3]. A typical client code looks like this:

```
c ← create_cursor(...)
while has_next(c) do
  x ← next(c)
  ...
```

In Java, the "for each" loop construct `for (E x: ...)` is nothing more than syntactic sugar for the above.

In this section we describe the use of predicates *enumerated* and *completed* to formally specify what is a cursor (Sec. 4.1), to verify a cursor implementation (Sec. 4.2), and to verify a client code that uses a cursor (Sec. 4.3).

[3] This is not mandatory. A cursor can be implemented as a persistent structure [6].

4.1 Cursor Specification

We assume two data types to be given: a type `elt` for the elements enumerated by the cursor, and a type `collection` for the collection whose elements are enumerated.

```
type elt
type collection
```

The term "collection" is to be taken broadly here. It does not necessarily designate a data structure but rather any data needed for the iteration specification. We model the cursor type as follows:

```
type cursor model {
          collection: collection;
  mutable    visited: seq elt;

}
```

The field `collection` is used to stock the collection of elements that is to be iterated by the cursor. The field `visited` contains the sequence of the elements enumerated by the cursor so far. This field is marked as mutable, to account for the imperative nature of the cursor. Finally, the `cursor` type is marked as being a *model type*. It means this is an abstract data type from the programming point of view. In particular, client code cannot access the `visited` field, preventing any modification of its contents. The specification, however, is free to refer to `cursor`'s field and typically will.

Next, we introduce the two predicates *enumerated* and *completed* to specify the cursor's behavior.

```
predicate enumerated (c: cursor) = ...
predicate completed  (c: cursor) = ...
```

Now we can provide suitable contracts to functions `has_next` and `next`. They are introduced as unimplemented functions with the keyword `val`.

```
val has_next (c: cursor) : bool =
  requires { enumerated c }
  ensures  { result ↔ not (completed c) }
```

In other words, function `has_next` decides whether the predicate `completed` holds. The second operation, `next`, is specified as follows:

```
val next (c: cursor) : elt
  requires { enumerated c }
  requires { not (completed c) }
  writes   { c }
  ensures  { enumerated c }
  ensures  { c.visited = snoc (old c.visited) result }
```

A call to `next` is only allowed when the iteration is not yet completed (the second `requires`). The postcondition guarantees that the returned element is appended

at the end of the `visited` sequence. This side effect is expressed with the `writes` clause.

The postcondition of `next` also guarantees that the `visited` sequence satisfies the `enumerated` predicate. Functions `has_next` and `next` also require predicate `enumerated` as a precondition. This is a way to ensure that the cursor remains in a consistent state. Suppose for instance that the cursor is enumerating the elements of an array. Nothing prevents us from mutating the array while the cursor is being used. If we do so, however, the `enumerated` predicate will not hold anymore and, consequently, we will not be able to call functions `has_next` and `next` anymore.

In practice, we also need to provide operations to create cursors. Such an operation looks as follows:

```
val create_cursor (t: collection) : cursor
  ensures { result.visited = empty }
  ensures { enumerated result }
  ensures { result.collection = t }
```

It returns a fresh cursor whose `visited` sequence is empty (first postcondition) and which is in a consistent state (second postcondition).

Collection modification. Taking an example of a cursor to traverse the elements of an array we can imagine the following code:

```
let c = create_cursor a in
a[0] ← 42;
let x = next c in
...
```

that modifies the array `a` after creating the cursor `c`. However, if we try to prove this program we will no be able to prove the precondition of function `next`, namely `coherent c`. The array has been modified and so has the cursor as it contains the array in the `collection` field.

4.2 Cursor Implementations

To validate our approach, we have implemented and verified several cursors using Why3. These examples include iterators for collections, such as arrays, and lists, and sets, as well as a symbol generator, the in-order traversal of a binary tree, the DFS traversal of a graph, and a cursor that merges the ordered sequences generated by two other cursors. For each cursor, we have

– refined the `cursor` data type, to add data specific fields. If we consider the cursor for an array, for instance, the refinement is as follows:

```
type cursor = { ghost mutable    visited: seq elt;
                      mutable      index: int;
                           collection: array elt; }
```

cursor	loc	los	time (sec)
gensym	12	30	0.03
array	12	23	0.05
list	15	28	0.40
set	12	22	13.74
binary tree	36	72	0.21
merge	36	75	2.83
dfs	48	85	11.02
total	171	335	

(a) Cursor Implementations

program	loc	los	time (sec)
array sum	8	12	0.70
list length	8	4	0.03
search	8	10	0.10
same fringe	36	72	0.21
check path	11	4	1.25
merge cursors	36	75	2.83
mjrty	32	22	1.67
total	139	199	

(b) Cursor Clients

Table 1: Experimental results.

– strengthen the `enumerated` predicate, so that it acts as a gluing invariant as well. For the array example, the gluing invariant adds the property that the `index` field is equal to the length of `visited`.

```
predicate enumerated (c: cursor) =
  (forall i. 0 ≤ i < length c.visited → c.visited[i] = c.array[i]) ∧
  c.index = length c.visited
```

– implemented and verified operations `next`, `has_next`, and `create_cursor`.

Table 1a shows the lines of code, the lines of specification (functions contracts, invariants, and auxiliary lemmas), and the total verification time (in seconds) for each cursor. All verification conditions are discharged automatically, using a combination of the SMT solvers Alt-Ergo, Z3, and CVC4.

4.3 Cursor Clients

We have also implemented and verified a number of client programs that make use of the cursors presented in the previous section. We do this in a modular way, *i.e.*, the client programs are only using the cursor interface (from Sec. 4.1) and have no access to the underlying implementation.

Our programs include summing the elements of an array, computing the length of a list, searching for a particular element in some abstract collection, solving the "same fringe" problem (comparing two binary trees using two in-order traversal cursors), checking for the existence of a path in a graph using a DFS cursor, merging two ordered sequences, and implementing Boyer & Moore's "mjrty" algorithm [2] using array cursors.

Table 1b shows the lines of code, the lines of specification, and the total verification time (in seconds) for each program. All verification conditions are discharged automatically. Source files are available online.

5 Higher-Order Iterators

In programming languages featuring first-class functions, iteration is commonly implemented as a higher-order function that takes as argument a function to be applied to each element of the enumerated sequence. In an imperative language, such a function can be as simple as

$$\texttt{iter} : (\mathit{elt} \to \mathit{unit}) \to \mathit{collection} \to \mathit{unit}$$

where *elt* is the type of the iteration elements, *collection* is the type of the collection to be iterated over, and *unit* is a type with no meaningful values. If the elements of a collection c are $x_1, \ldots, x_n$, in that order, then a call to `iter` f c simply amounts to evaluate $f(x_1), \ldots, f(x_n)$ sequentially. Assuming the elements of c are integers, we can sum them using

$$s \leftarrow 0;\ \texttt{iter}\ (\lambda x.\, s \leftarrow s + x)\ c$$

where λ introduces an anonymous function. The recent introduction of closures in languages such as C++ and Java eases this style of programming.

Higher-order iterators coexist with cursors, allowing the user to choose the paradigm that suits best. The main difference between the two is that control is given to the producer in the case of a higher-order iterator, while it is given to the consumer in the case of a cursor.

In this section we describe a methodology to specify and verify higher-order iterators using *enumerated* and *completed* predicates. As we did for cursors, we intend to verify both implementations of `iter` functions (Sec. 5.1), and client code using `iter` functions (Sec. 5.2). One way to tackle the verification of higher-order functions is to use a higher-order (program) logic, in such a way that we can quantify over the specification of function arguments. There exist already several systems in which we can do so; we will discuss those in Sec. 6. We consider here a different approach, which only requires first-order logic. This is possible thanks to the *abstraction barrier* provided by the *enumerated*/*completed* predicates. On both sides of this interface, we are making distinct first-order program proofs, one for the implementation of `iter` and one for each call to `iter`.

Currently, Why3 does not support the use of effectful higher-order code. To circumvent this limitation, we have developed a prototype tool that reads both implementations and uses of higher-order iterators, together with specification and possible annotations, and turns them into regular Why3 programs to be verified.

5.1 Verifying an Iterator

Given an implementation of some `iter` function, our approach consists in automatically building a first-order function `iter_correct` whose correctness implies that of `iter`. Once function `iter_correct` is verified, we do not need it anymore.

We obtain function `iter_correct` by specializing the code of `iter` for a particular function that appends the element it receives (the next element of the

iteration) to a sequence stored in a global variable `visited`. Then we can verify the resulting code against the specification given by the predicates *enumerated* and *completed*.

Let us consider the case of the in-order traversal of a binary tree, the type of which is:

```
type tree = E | N tree elt tree
```

In our prototype, we implement in-order traversal as follows:

```
let rec iter (f: elt → unit) (t: tree) : unit
  with { enumerated (visited, t) = ...
         completed  (visited, t) = ... }
= match t with
  | E       → ()
  | N l x r → iter f l; f x; iter f r
  end
```

The iteration specification is introduced with the keyword `with`, as the pair of the two predicates `enumerated` and `completed` (whose definition is omitted here). In this case, `iter` is defined recursively, as it is the simplest way to do. Yet this is not mandatory. Our technique applies as well to iterative implementations. From this definition, we automatically generate the following Why3 function `iter_correct`, together with its specification.

```
val visited: ref (seq elt)

let iter_correct (t0: tree)
  requires { !visited == empty ∧ enumerated (empty, t0) }
  ensures  { enumerated (!visited, t0) ∧ completed (!visited, t0) }
= let f x = visited := snoc !visited x in
  let rec iter0 (t: tree) : unit =
    match t with
    | E       → ()
    | N l x r → iter0 l; f x; iter0 r
    end in
  iter0 t0
```

This function takes a tree `t0` as argument. It stands for the original argument of `iter`. The specification expresses that if we start with an empty `visited` sequence, then we end up with a completed iteration for the tree `t0`. To verify this code, we have to equip function `iter0` with suitable annotations. This part is not done automatically, as it depends on the implementation of `iter` and the nature of *enumerated* and *completed*.

Using our prototype tool, we have verified several implementations of iterators, including traversals of arrays, lists, trees, and abstract collections. The resulting verification conditions are all discharged automatically by SMT solvers.

5.2 Using an Iterator

As we did for the implementation of the `iter` function, we propose a methodology to verify a client code using `iter` by translating it to a first-order program.

Predicates *enumerated* and *completed* are used to specify the iteration, and the client code has no access to the implementation of `iter`. Our idea is to transform the client code by replacing the use of `iter` with a `while` loop that uses a cursor. This cursor is specified exactly as in Sec. 4.1. Once again abstraction is the key: the client code only relies on the iteration specification, and not on the way it is implemented.

Let us illustrate the idea on an example. We consider a program that takes a list as an argument and returns a list containing the same elements without repetitions. It uses an `iter` function to traverse the input list and a hash table to store the elements we have seen so far. Considering the following type for lists

```
type list = Nil | Cons elt list
```

we can use our prototype to define the following client program (assuming hash table operations provided by a module `H`):

```
let uniq (l: list) : list
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  iter (fun x →
         if not (H.mem x h) then begin H.add x h; r := Cons x !r end)
       l;
  !r
```

The code first declares a new hash table `h` and a reference `r` to hold the output list. The consumer function checks, each time it is called, whether the element `x` is not yet in the hash table. If so, it adds `x` both to the table `h` and to the list `r`. When the iteration completes, we return the contents of `r`.

To verify function `uniq`, we need to equip the iteration with a suitable "loop" invariant. We use here the term "loop" in a loose way, to refer to the iteration performed by the `iter` function. To allow this invariant to refer to the sequence of already enumerated elements, we add an extra ghost argument `v` to the consumer function. The code now looks as follows:

```
iter (fun (ghost v) x →
        invariant { ...user loop invariant... }
        if not (H.mem x h) then begin H.add x h; r := Cons x !r end)
```

For this program, a suitable invariant is the following:

```
distinct !r ∧ (forall x. mem x v ↔ mem x !r) ∧
(forall x. H.contains h x ↔ mem x !r)
```

It states that at each step of the iteration the accumulator `r` contains exactly the elements enumerated so far, without repetition. The last clause states that the elements in the hash table are exactly the elements of `r`.

We now turn the `uniq` program into a first-order implementation `uniq_correct` that uses a cursor to perform the same iteration as the `iter` function. The following is the Why3 code resulting from this transformation (as produced automatically by our prototype tool):

```
let uniq_correct (l: list elt) : list elt
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  let _c = create_cursor l in
  while has_next _c t do
    invariant { enumerated _c }
    invariant { let v = _c.visited in ...user loop invariant... }
    let x = next _c l in
    if not (H.mem x h) then begin H.add x h; r := Cons x !r end
  done;
  !r
```

The invariant `enumerated _c` is automatically added to the loop[4]. The second invariant is the one that was given by the user, where `v` is bound to the sequence contained in the cursor. The cursor functions `create_cursor`, `next`, and `has_next` are given the same contracts as in Sec. 4.1. The body of the consumer function is turned into the loop body, and `x` is bound to the next iteration element, as returned by `next`.

Then we can feed the program `uniq_correct` to Why3 for verification. In this case, all verification conditions are discharged automatically by SMT solvers. This implies that the original higher-order `uniq` program is correct with respect to its specification, provided function `iter` is implemented and proved correct w.r.t. the same *enumerated*/*completed* specification. The latter can be done using the technique presented in the previous section.

It is worth pointing out that the consumer function passed to `iter` is free to have side effects. (In the example above, it does, as it fills the hash table.) In particular, it could jeopardize the iteration by mutating data on which the iteration relies. This is not an issue, though, since we have to prove the preservation of the loop invariant `enumerated`. In this respect, the situation is not different from the use of cursors, as described in Sec. 4.1.

6 Related Work

The idea of formally specifying and proving cursors is not new. Weide presents a formal specification for the cursors' behavior [16] using the *RESOLVE* language [10]. A collection is modeled as a finite set (in the mathematical sense) and a cursor is specified using a `past` sequence corresponding to our `visited` and another `future` sequence corresponding to remaining elements. A third sequence, `original`, contains the set of elements of the collection. Under such formalization, a cursor can only be used with finite collections and the traversal is necessarily deterministic. The author also presents a mechanism to ensure coherence, by

[4] Here we use a definition of `enumerated` that takes as argument a cursor. This can be easily derived from the definition of `enumerated` that was given to specify the iterator.

means of extra operations over cursors, `Start_Iterator` and `Finish_Iterator`, that should limit all the cursor uses. In this way, and contrary to our approach, the validity of a cursor can only be verified once the traversal is finished.

In the literature we can find many cursors formalization and proof examples under the more general context of data structures library verification. One example is that of the EiffelBase2 library [12], a container library for the Eiffel language. The verification task is performed using the AutoProof system. However, EiffelBase2 offers no generic presentation of cursors.

Many tools exist that tackle the verification of higher-order effectful programs, in particular of higher-order iterators. These are normally based on rich specification logics and type systems. Liquid Types [13] is a type system with refinement types extracted from a decidable logic. This type system is used to infer simple "loop invariants" from a given code. In our case, the user supplies the loop invariant and, contrary to the Liquid Types approach, we apply and prove an iterator client without access to the iterator implementation, in a modular way.

Vazou *et al* [15] present a technique to verify a call to a `foldr` function (another iterator, very close to `iter`) over lists. This technique consists in annotating the program with a dependent type that expresses an invariant about the list of already processed elements. We provide a similar invariant when calling an `iter` function. The main difference is that our approach is not limited to lists: using predicates *enumerated* and *completed* we can specify many kinds of iteration.

Dependent types and monad structures are used in the F* tool [14] as the theoretical basis to tackle the proof of higher-order programs with effects. F* can be used both as a programming language and as a proof assistant, featuring a higher-order specification and programming language. This tool has been used to verify many complex effectful programs including cryptographic protocols and the mechanization of lambda calculi metatheory. Even though F* is able to use SMT solvers during the proving process, it seems that the verification of nontrivial (effectful) higher-order programs is out of the realm of automatic provers. In particular, the specification of a higher-order iterator is very similar to what one would write in a general-purpose proof assistant like Coq.

The CFML tool [3] uses characteristic formulas to verify OCaml code within the Coq proof assistant. Characteristic formula is a higher-order formula that can be generated from a source code and its specification, and that describes the semantics of a given program. Using a proof assistant based on higher-order logic, the characteristic formula can be exploited to prove complex properties about that program. Up to now, CFML has been used to verify several nontrivial higher-order imperative programs, including higher-order iterators over mutable data structures. However, the specification used to describe a higher-order iterator is always tied to a specific collection data type.

Ynot [11] is a library for the Coq proof assistant that can be used to write and verify imperative programs. It is based on Hoare Type Theory and the use of monads and separation logic to reason about effects. An implementation of

imperative finite maps has been verified with Ynot, including a *fold*-like (effectful) iterator. The theoretical techniques employed by Ynot seem to make difficult its use in an automatic proof process.

7 Conclusion and Perspectives

In this paper we presented an approach to specify programs performing iterations. Our proposal consists in specifying two predicates *enumerated* and *completed* characterizing the sequence of already enumerated elements. Our specification allows, notably, non-deterministic and infinite iterations. This approach can be applied to different iteration paradigms.

To validate our idea, we applied it to the specification of two particular forms of iteration, namely cursors and higher-order iterators. We wrote several examples of iterators and client codes for each paradigm. Using the Why3 deductive verification tool we were able to formally prove that these implementations are correct. It is worth noting that our approach to specify an iteration via predicates `enumerated` and `completed` is not tied to Why3. Any other deductive verification tool could be used instead.

To verify higher-order iterators, we proposed a mechanical translation of a higher-order code (either an iterator implementation or a client code) into a first-order program. The specification of this first-order program is automatically derived from the predicates *enumerated* and *completed*, and the correctness of the generated code implies that of the initial higher-order code.

Perspectives. On a short term perspective, we intend to extend Why3 with a `for` loop à la Java based on cursors. This will be of particular interest for a longer-term project of verifying a realistic graph library with Why3. Indeed, graph algorithms heavily rely on the use of iterators, for example to traverse vertices of a graph or neighbors of a vertex. It remains to show that our specification of iteration is well suited for the verification of such algorithms, particularly in a context where we seek proofs as most automatic as possible.

Besides, we think that our proposal could apply as well to other iteration paradigms, such as streams (implemented as lazy lists) or generators (implemented as coroutines). We intend to explore this question in the future.

Acknowledgments. We thank Clément Fumex, Chantal Keller, Claude Marché, Andrei Paskevich, Vitor Pereira, François Pottier, and Simão Melo de Sousa for their comments on earlier versions of this paper.

References

1. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) 17(6), 709–727 (2015)

2. Boyer, R.S., Moore, J.S.: Mjrty: A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. pp. 105–118 (1991)
3. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP). pp. 418–430. ACM, Tokyo, Japan (September 2011)
4. Clochard, M., Filliâtre, J.C., Marché, C., Paskevich, A.: Formalizing semantics with an automatic program verifier. In: Giannakopoulou, D., Kroening, D. (eds.) 6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE). LNCS, vol. 8471, pp. 37–51. Springer, Vienna, Austria (Jul 2014)
5. Coplien, J.O.: Advanced C++ Programming Styles and Idioms. Addison-Wesley (1992)
6. Filliâtre, J.C.: Backtracking iterators. In: ACM SIGPLAN Workshop on ML. Portland, Oregon (Sep 2006)
7. Filliâtre, J.C.: One logic to use them all. In: 24th International Conference on Automated Deduction (CADE-24). Lecture Notes in Artificial Intelligence, vol. 7898, pp. 1–20. Springer, Lake Placid, USA (June 2013)
8. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In: Biere, A., Bloem, R. (eds.) 26th International Conference on Computer Aided Verification. LNCS, vol. 8859, pp. 1–16. Springer, Vienna, Austria (Jul 2014)
9. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. LNCS, vol. 7792, pp. 125–128. Springer (Mar 2013)
10. Kulczycki, G., Sitaraman, M., Krone, J., Hollingsworth, J.E., Ogden, W.F., Weide, B.W., Bucci, P., Cook, C.T., Drachova-Strang, S., Durkee, B., Harton, H.K., Heym, W.D., Hoffman, D., Smith, H., Sun, Y., Tagore, A., Yasmin, N., Zaccai, D.: A language for building verified software components. In: Favaro, J.M., Morisio, M. (eds.) Safe and Secure Software Reuse. ICSR 2013. LNCS, vol. 7925, pp. 308–314. Springer (Jun 2013)
11. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: Proceedings of ICFP'08 (2008)
12. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: Bjørner, N., de Boer, F.D. (eds.) FM 2015, Oslo, Norway, June 24-26, 2015. LNCS, vol. 9109, pp. 414–434. Springer (2015)
13. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI 2008, Tucson, AZ, USA, June 7-13, 2008. pp. 159–169. ACM (2008)
14. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: 43rd ACM Symposium on Principles of Programming Languages (POPL). pp. 256–270. ACM (Jan 2016)
15. Vazou, N., Rondon, P., Jhala, R.: Abstract Refinement Types. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems, LNCS, vol. 7792, pp. 209–228. Springer Berlin Heidelberg (2013)
16. Weide, B.W.: SAVCBS 2006 challenge: Specification of iterators. In: Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems. pp. 75–77. SAVCBS '06, ACM, New York, NY, USA (2006)